

7N-61-TM

C1325B



National Aeronautics and
Space Administration

Ames Research Center
Moffett Field, California 94035

ARC 275 (Rev Feb 81)

Intercube Communication for the iPSC/860

E. Barszcz*

Report Number: RNR-91-030

29 October 1991

Abstract

In this paper, new functions that enable efficient intercube communication on the Intel iPSC/860 are introduced. Communication between multiple cubes (power-of-two number of processor nodes) within the Intel iPSC/860 is a desirable feature to facilitate the implementation of interdisciplinary problems such as the grand challenge problems of the High Performance Computing and Communications Project (HPCCP). Intercube communication allows programs for each discipline to be developed independently on the hypercube and then integrated at the interface boundaries using intercube communication.

*NAS Applied Research Branch, NASA Ames Research Center, Moffett Field, CA 94035.

1 Introduction

In this paper, new functions that enable efficient intercube communication on the Intel iPSC/860 are introduced. Communication between multiple cubes (power-of-two number of processor nodes) within the Intel iPSC/860 is a desirable feature to facilitate the implementation of interdisciplinary problems such as the grand challenge problems of the High Performance Computing and Communications Project (HPCCP). Intercube communication allows programs for each discipline to be developed independently on the hypercube and then integrated at the interface boundaries using intercube communication. Intercube communication is also useful within a single discipline where the physical domain is broken into several zones (computational domains) to facilitate grid generation or to accommodate bodies in relative motion. For good load balance, in both interdisciplinary and multizone problems, the number of computational nodes assigned to a computational domain should match the workload associated with that domain.

Currently, there are three ways to implement intercube communication on the Intel iPSC/860: individual cubes can communicate through the service resource module (SRM), via a shared file on the concurrent file system (CFS), or allocate a cube large enough to hold all desired subcubes and manage subcube allocation and communication from within the user program. All three methods have problems. Communicating through the SRM is slow because the SRM must receive the message, attach to the destination cube and then forward the message. Communicating through the CFS is not much better, since cubes must coordinate access to shared files. Having the user allocate and partition a single large cube places the burden of partitioning on the user and makes it more difficult to do independent development of domain specific code. Also, a single large cube partitioned into subcubes may lead to poor load balance if the code for an domain is designed to work with a power-of-2 number of processor nodes and the sum over all domains is not a power-of-2 number of nodes.

The intercube communication described in this paper is fast and efficient. It is fast because it uses the hypercube wires allowing messages to be sent in parallel with no bottleneck. It is efficient because there is little overhead associated with sending an intercube message. Overhead is primarily due to the verification and validation of the destination and contention for the wires from other users communicating with the SRM or CFS.

In addition to interdisciplinary and multizone programs, intercube communication is useful for functional partitioning of programs. For example, if one cube is generating a large amount of data, a second cube can be processing the data as it is generated in parallel.

In the remainder of the paper a discussion dealing with routing, allocation and contention is followed by the implementation of the intercube communication and an example demonstrating intercube communication. It is assumed that the reader is familiar with the Intel iPSC/860 system calls. For more information about the iPSC/860 see references [1] and [2].

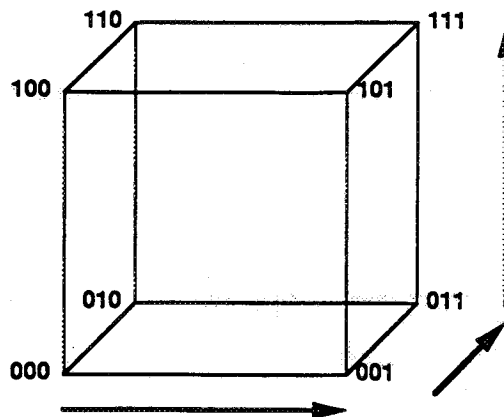


Figure 1: e-cube routing of a message from node 0 to node 7.

2 Routing, Allocation and Contention

In this section, e-cube routing, cube allocation and sources of intercube communication contention are described followed by a discussion of when intercube communication may cause intracube communication contention within another cube.

2.1 e-cube Routing

The routing algorithm used on the Intel hypercubes is called e-cube routing. In e-cube routing, the source address and destination address are XORed and the resulting nonzero entries indicate which wires the message will traverse. Wires are traversed starting from the lowest order nonzero and proceed to the highest order nonzero. In practice, as the message is passed to each new node, the destination address is XORed with the local address and the message moved across the wire indicated by lowest order nonzero. When XORing with the local address, if the result has no nonzero entries then the message has reached its destination. Figure 1 shows the path a message takes from node 0 to node 7 using e-cube routing.

2.2 Cube Allocation

When allocating a cube on an Intel hypercube, the physical address of logical node zero must be a multiple of the cube size. The operating system will not allocate a cube starting at a node whose physical address is not a multiple of the cube size.

Also, if a location is specified when a cube is requested, nodes with contiguous physical addresses will be assigned. If contiguous nodes cannot be assigned, the cube will not be allocated.

If a location is not specified, the operating system will allocate a cube if possible. However, the nodes are not guaranteed to be contiguous. The operating system tries to allocate contiguous nodes and then looks for other hypercubes if there are not enough contiguous nodes. Figure 2a shows a cube of size four formed from contiguous

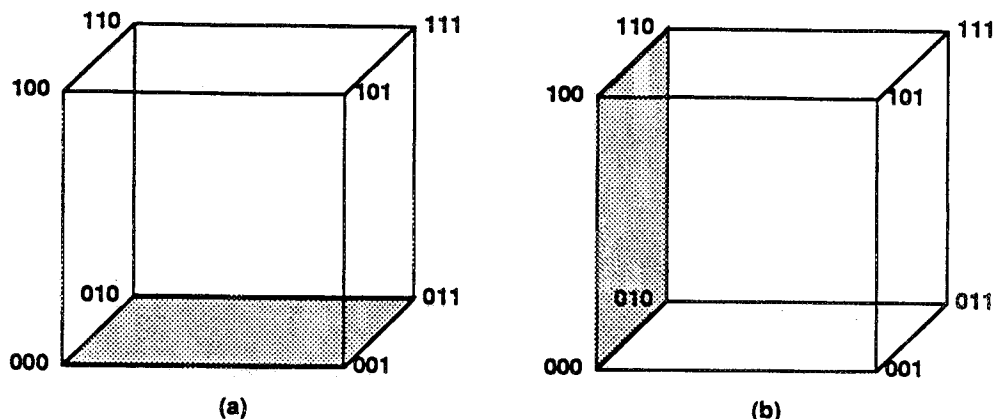


Figure 2: Cubes of size 4 formed from (a) contiguous nodes and (b) noncontiguous nodes.

nodes whereas Figure 2b shows a cube of size four formed from noncontiguous nodes. Contiguous node allocation is the most common case [3].

2.3 Intercube Contention

Possible sources of contention for intercube communication are the operating system downloading programs, users communicating with the SRM, users communicating with the concurrent file system (CFS). All of these sources exist and have existed within current systems and the author is not aware of users complaining about them.

One characteristic of all intercube communication is that it occurs infrequently. Programs are downloaded once per cube. Programs are I/O bound if they communicate with the CFS to frequently. The SRM is a bottleneck if users communicate with it to often. Communicating interface boundary data in interdisciplinary or multizone programs also occurs infrequently.

2.4 Intracube Contention

During intercube communication it is possible to interfere with intracube communication occurring within another cube. However, if all cubes are formed from contiguously allocated nodes with starting addresses that are a multiple of the cube size, intercube communication cannot interfere with intracube communication within another cube.

The potential for intercube communication to interfere with intracube communication within another cube is demonstrated in Figure 3. In Figure 3 there are two users, user 1 owns two cubes each of size one (node 0 and node 7) and user 2 owns one cube of size two (nodes 1 and 3). The cube owned by user 2 is not formed from contiguous nodes creating the possibility of contention. When user 1 sends a message from cube 1 to cube 2 (node 0 to node 7), the e-cube routing algorithm requires it to pass between node 1 and node 3 potentially interfering with the intracube communication of user 2.

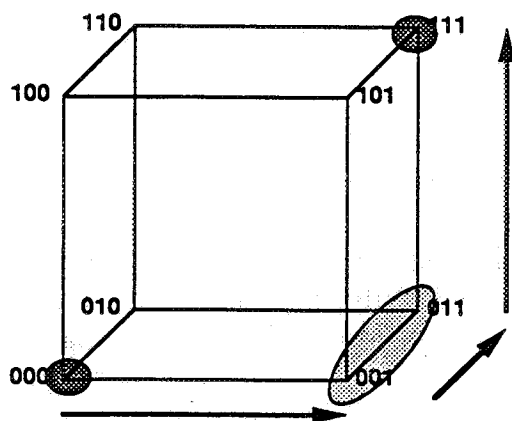


Figure 3: Intracube contention from intercube communication.

Fortunately, when all cubes are allocated from contiguous nodes starting on physical node addresses that are a multiple of the cube size, intercube communication cannot interfere with the intracube communication within another cube. To see this, let the cube size be given by N and the cube dimension by $D = \log_2(N)$. Since the starting physical node address must be a multiple of the cube size, the low order D bits of the address are zero. The N contiguous physical addresses of a cube use the low order D bits. When sending an intercube message, any bit differences between the source and destination in the low order D bits are eliminated by sending the message around inside the local cube¹. At this point, only high order bit differences are left. Every cube has a natural partner formed from the cube of the same size and starting address but with the $(D + 1)$ low order bit complemented². They are natural partners because they are the only cubes of size N in the proper locations to form a cube of size $2N$ satisfying the contiguous address and starting address constraints. If the $(D + 1)$ bit differs between the source and destination, the message will travel between the two cubes of size N . However, this link cannot be in use by either cube for intracube communication and so does not cause interference inside the other cube. By induction it can be shown that given contiguous nodes and starting addresses that are a multiple of the cube size, intercube communication cannot interfere with intracube communication within another cube.

3 Implementation

The major premise for the intercube communication implementation is that the number of cubes and size of cubes required is known and does not change. Static allocation of cubes is a valid model for any application that has multiple fixed sized computational domains that need to communicate.

To make intercube communication flexible and easy to support, the intercube communication mechanism is general and implemented using a small number of new

¹Recall that e-cube routing processes bit differences from low order to high order.

²A cube equal to the whole machine has no partner.

routines. Having a general intercube communication mechanism allows applications to tailor intercube communication to individual needs. Given a small number of new routines, less support is required when the operating system or underlying hardware changes.

This implementation of intercube communication requires a host program and each cube participating in intercube communication to be *registered*. After all cubes have been registered and before any communication takes place, information about each cube and the nodes within each cube is downloaded to all participating nodes. Then to perform intercube communication, a (cube name, node number) pair is mapped into a system node address that is passed to the regular communication routines such as `csend`. This implies that forced message types [1] and pairwise exchange [5] will work with intercube communication. Currently, there is a system of 10 cubes that can be allocated³.

3.1 Data Structure

The data structure containing information on all cubes registered has the following fields:

- Local cube name
- Total number of cubes
- For each cube
 - Cubename
 - Number of nodes
 - Mapping function from logical node numbers to system node addresses

For the Intel iPSC/860, with 128 nodes, this data structure is less than 1 KB⁴ and is directly proportional to the total number of nodes allocated. It is felt that 1 KB is small enough that each node has a copy of the data structure.

To index into the table, a hash function based on the cube name is used. The hash function groups the cube name into blocks of four characters (treating a block as an unsigned integer), adds them together ignoring overflow and returns the sum modulo the table size. If the last group has less than four characters, the group is right justified and then added.

If a collision occurs (the table entry indicated by the hash function is not empty and not the desired entry), entries on both sides of the hash location are examined. If collisions occur again, a linear search is started at the hash location plus two. An empty slot is guaranteed for insertion because the table size matches the maximum number of cubes that can be allocated. When searching for an entry in the table, if an empty slot is encountered or the whole table has been searched without finding the desired entry, the cube name is invalid.

³Only nine cubes can be allocated if you have a CFS since the CFS requires a cube.

⁴This could be compressed significantly at the cost of extra processing to calculate the system node addresses.

3.2 New Host Routines

Functional descriptions of the newly implemented host routines are given below and the actual interfaces to FORTRAN and C are in appendices A and B respectively. Each cube must be given a unique cube name up to 15 characters in length.

initcubecomm()

Initcubecomm freezes the number of cubes that are participating in intercube communication, interrogates the operating system for the system node addresses, fills in the data structure and then downloads it to all participating nodes. It must be called before performing any communication.

setactivecube(cube_name)

Setactivecube takes a cube name (up to 15 characters) as an argument, verifies that the cube is owned by the user, if it is the first instance of the cube name, registers the cube, and then does an **attach** and **setpid** for that cube. **Setactivecube** is called every time the the host wants to communicates with a different cube. **Sethostpid** must be called before the first call to **setactivecube**.

sethostpid(pid)

Sethostpid records a single host process identifier that is used for all cubes. The **setpid** function is called by **setactivecube** after attaching to a cube. **Sethostpid** must be called once before the first call to **setactivecube**.

3.3 New Node Routines

Functional descriptions of the newly implemented node routines are given below and the actual interfaces to FORTRAN and C are in appendices A and B respectively.

cmpname(cube_name1, cube_name2)

Cmpname is a logical function that takes two cube names as arguments and returns TRUE if the names match and FALSE if they don't match.

cubemap(cube_name, node)

Cubemap takes a cube name and logical node number as input parameters and returns the system node address. The system node address can then be passed to either the **csend** or **isend** communication routines⁵. Before returning the system node address, **cubemap** verifies that the cube is owned by the user and checks that the node number is valid for that cube.

cubesize(cube_name)

Cubesize returns the size of the specified cube.

initcubecomm()

Initcubecomm downloads the data structure containing information on all participating nodes. It must be called before performing any communication.

⁵Existing communication routines recognize system node addresses as well as logical node addresses so no new communication primitives are required.

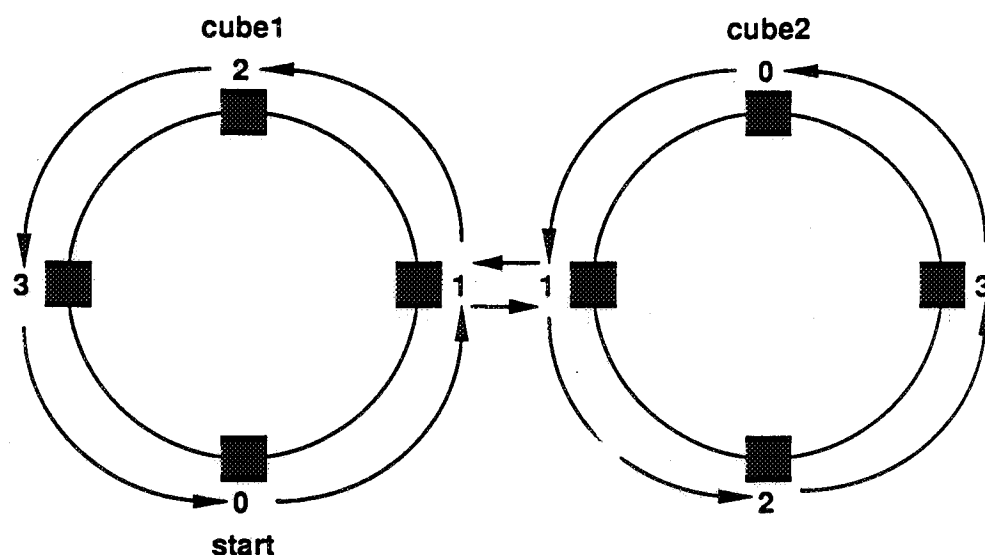


Figure 4: Ring formed from two cubes of size four.

mycube()

Mycube returns the local cube name.

numcubes()

Numcubes returns the number of cubes willing to participate in intercube communication.

4 Example

As an example of intercube communication, the sample ring code provided by Intel is modified to send a message around a ring formed from two different cubes which are joined at logical node one of each cube. Figure 4 shows two cubes of size four linked together by logical node 1 in each cube to form a single ring. To start a message around the ring, cubel receives a message from the host describing the length of the message and the number of times it should go around the ring. After each lap cubel sends the lap count back to the host. When the message has gone around the required number of times cubel reports the elapsed time.

Changes made to the host and node programs are shown in appendices C and D respectively.

4.1 Results

To determine whether the cubemap function has significant overhead, the ring example is timed with two cubes of size eight and compared to the ring program using a single cube of size sixteen. It should be noted that cubemap can be called once and the destination system node address saved, amortizing the overhead across all laps. Cubemap is called every lap to see if it has significant overhead.

There is no difference in time when sending a 32 byte message 20 times around the ring formed from two cubes, each of size eight, when compared to sending the same size message around a single cube of size sixteen⁶. The location in decimal and binary of the two cubes is given in Table 1. The starting address expressed in binary demonstrate that intercube communication takes place since the addresses differ in all high order bits.

Cube Size	Location	Binary
8	0	0000000
8	120	1111000

Table 1: Cube sizes and locations.

A closer examination of the overhead indicates the communication with the SRM is the determining factor in the elapsed time. Timings using `dlock()` calls placed around `cubemap` in the node program reveal a cost of 40 microseconds per call to `cubemap`. Of this 40 microseconds, 18 microseconds is associated with copying the FORTRAN string into a C string and setting an end-of-string marker and 22 microseconds to look up the entry in the table.

5 Summary

Efficient general intercube communication has been implemented using a small number of new functions. By keeping the number of functions small, any changes to the operating system or machine can be accommodated easily. By making the routines general, an application can tailor intercube communication to its needs.

The number and size of cubes involved in intercube communication is static. Cubes desiring intercube communication must register and receive a data structure containing information about all participating cubes before performing any communication. The static allocation model applies to any application that has multiple fixed sized computational domains or can be functionally partitioned.

The intercube communication routines work from an Iris workstation as remote host and are callable from either FORTRAN or C. The routines should work on an Intel iPSC/2 and it is anticipated that they will work on the Intel Delta machine at CalTech and possibly on the Intel Sigma machine.

Acknowledgements

The author would like to thank Paul Pierce of Intel SSD for discussions and information about Intel system routines and Sisira Weeratunga of Computer Sciences Corporation for discussions and beta testing the routines. Sisira is currently using the routines to implement an interdisciplinary code on an iPSC/860.

References

⁶This version of the ring program is not Gray coded.

1. "iPSC/2 and iPSC/860 Programmer's Reference Manual", *Intel Supercomputer Systems Division, Beaverton, Oregon*, April 1991.
2. "iPSC/2 and iPSC/860 User's Guide", *Intel Supercomputer Systems Division, Beaverton, Oregon*, June 1990.
3. Private communication with Paul Pierce of Intel Scientific Computers Division.
4. Seidel, S., Lee, M.H., and Fotedar, S., "Concurrent Bidirectional Communication on the Intel iPSC/860 and iPSC/2", *Michigan Technological University Technical Report CS-TR 90-06* (November 1990).

Appendix A: FORTRAN Interface

The FORTRAN interface is presented in style similiar to the one used in the iPSC/2 and iPSC/860 Programmer's Reference Manual [1].

Synopsis

LOGICAL FUNCTION COMPARE(*cubename1*, *cubename2*)

Parameter Declarations

CHARACTER *cubename1**(*)
CHARACTER *cubename2**(*)

Return Value

TRUE if *cubename1* matches *cubename2*, FALSE otherwise.

Environment

Node

Discussion

Does not check for valid cube names.

Errors

None

Synopsis

INTEGER FUNCTION CUBEMAP(*cubename*, *node*)

Parameter Declarations

CHARACTER *cubename**(*)
INTEGER *node*

Return Value

System node address for the node specified by (*cubename*, *node*).

Environment

Node

Discussion

Cubemap takes a cube name and logical node number as input parameters and returns the system node address. The system node address can then be passed to either the **csend** or **isend** communication routines. Before returning the system node address, **cubemap** verifies that the cube is owned by the user and checks that the node number is valid for that cube.

Errors

cubemap: Invalid Cube Name

Use a valid cube name. Program will hang until killed by user.

cubemap: Invalid Node

Use a valid node number. Program will hang until killed by user.

Synopsis

INTEGER FUNCTION **CUBESIZE**(*cubename*)

Parameter Declarations

CHARACTER *cubename**(*)

Return Value

Number of nodes in the cube specified by *cubename*.

Environment

Node

Discussion

May be called after **initcubecomm**.

Errors

cuBemap: Invalid Cube Name

Use a valid cube name. Program will hang until killed by user.

Synopsis

SUBROUTINE INITCUBECOMM()

Parameter Declarations

None

Return Value

None

Environment

Host, Node

Discussion

On the host, `initcubecomm` *freezes* the number of cubes that are participating in intercube communication, interrogates the operating system for the system node addresses, fills the data structure and then downloads it to all participating nodes.

On the nodes, it receives the data structure from the host.

`Initcubecomm` *must* be called on the host and nodes before performing any communication.

Errors

None

Synopsis

CHARACTER*15 FUNCTION MYCUBE()

Parameter Declarations

None

Return Value

Character string containing the name of the local cube.

Environment

Node

Discussion

May be called after `initcubecomm`.

Errors

None.

Synopsis

INTEGER FUNCTION NUMCUBES()

Parameter Declarations

None

Return Value

Number of cubes participating in intercube communication.

Environment

Node

Discussion

May be called after `initcubecomfn`.

Errors

None.

Synopsis

SUBROUTINE SETACTIVECUBE(*cubename*)

Parameter Declarations

CHARACTER *cubename**(*)

Return Value

None

Environment

Host

Discussion

Setactivecube takes a cube name as an argument, verifies that the cube is owned by the user, if it is the first instance of the cube name, registers the cube, and then does an **attach** and **setpid** for that cube. **Setactivecube** is called every time the the host wants to communicates with a different cube.

Sethostpid *must* be called before the first call to **setactivecube**.

Errors

setactivecube: Invalid Cube Name

Use a valid cube name. Program will exit.

Synopsis

SUBROUTINE **SETHOSTPID**(*pid*)

Parameter Declarations

INTEGER *pid*

Return Value

None

Environment

Host

Discussion

Sethostpid records a single host process identifier that is used for all cubes. The **setpid** function is called by **setactivecube** after attaching to a cube.

Sethostpid *must* be called once before the first call to **setactivecube**.

Errors

sethostpid: Already Set Host PID

Use **sethostpid** once at beginning of host program. Program will exit.

Appendix B: C Interface

The C interface is presented in style similiar to the one used in the iPSC/2 and iPSC/860 Programmer's Reference Manual [1].

Synopsis

```
long cmpname(cubename1, cubename2)
```

Parameter Declarations

```
char *cubename1  
char *cubename2
```

Return Value

1 if *cubename1* matches *cubename2*, 0 otherwise.

Environment

Node

Discussion

Does not check for valid cube names.

Errors

None

Synopsis

long **cubemap**(*cubename*, *node*)

Parameter Declarations

char **cubename*
long *node*

Return Value

System node address for the node specified by (*cubename*, *node*).

c

Environment

Node

Discussion

Cubemap takes a cube name and logical node number as input parameters and returns the system node address. The system node address can then be passed to either the **csend** or **isend** communication routines. Before returning the system node address, **cubemap** verifies that the cube is owned by the user and checks that the node number is valid for that cube.

Errors

cubemap: Invalid Cube Name

Use a valid cube name. Program will hang until killed by user.

cubemap: Invalid Node

Use a valid node number. Program will hang until killed by user.

Synopsis

`long cubesize(cubename)`

Parameter Declarations

`char *cubename`

Return Value

Number of nodes in the cube specified by *cubename*.

Environment

Node

Discussion

May be called after `initcubecomim`.

Errors

`cubemap`: Invalid Cube Name

Use a valid cube name. Program will hang until killed by user.

Synopsis

initcubecomm()

Parameter Declarations

None

Return Value

None

Environment

Host, Node

Discussion

On the host, **initcubecomm** *freezes* the number of cubes that are participating in intercube communication, interrogates the operating system for the system node addresses, fills the data structure and then downloads it to all participating nodes.

On the nodes, it receives the data structure from the host.

Initcubecomm *must* be called on the host and nodes before performing any communication.

Errors

None

Synopsis

`char *mycube()`

Parameter Declarations

None

Return Value

Character string containing the name of the local cube.

Environment

Node

Discussion

May be called after `initcubecomm`.

Errors

None.

Synopsis

`long numcubes()`

Parameter Declarations

None

Return Value

Number of cubes participating in intercube communication.

Environment

Node

Discussion

May be called after `initcubecomm`.

Errors

None.

Synopsis

setactivecube(*cubename*)

Parameter Declarations

char **cubename*

Return Value

None

Environment

Host

Discussion

Setactivecube takes a cube name as an argument, verifies that the cube is owned by the user, if it is the first instance of the cube name, registers the cube, and then does an **attach** and **setpid** for that cube. **Setactivecube** is called every time the the host wants to communicates with a different cube.

Sethostpid *must* be called before the first call to **setactivecube**. ...

Errors

setactivecube: Invalid Cube Name

Use a valid cube name. Program will exit.

Synopsis

sethostpid(*pid*)

Parameter Declarations

long *pid*

Return Value

None

Environment^c

Host

Discussion

Sethostpid records a single host process identifier that is used for all cubes. The **setpid** function is called by **setactivecube** after attaching to a cube.

Sethostpid *must* be called once before the first call to **setactivecube**.

Errors

sethostpid: Already Set Host PID

Use **sethostpid** once at beginning of host program. Program will exit.

Appendix C: FORTRAN Ring Example Host Code Modifications

```
parameter ( NUMCUBES = 2 )
character*15 cube(NUMCUBES)
character*1 digit(0:9)
data digit/'0','1',...,'9'/
:
do i = 1, NUMCUBES
    cube(i) = "cube" // digit(i)
    call getcube( cube(i),...)
end do
call sethostpid( pid )
do i = 1, NUMCUBES
    call setactivecube( cube(i) )
    call load( 'node', ALLNODES, NODEPID )
end do
call initcubecomm()
:
call setactivecube( cube(1) )
call csend( INITTYPE, msgbuf, MSGSIZE, 0, NODEPID )
:
call crecv( COUNTTYPE, msgbuf, CNTMSGSIZE )
:
do i = 1, NUMCUBES
    call setactivecube( cube(i) )
    call killcube( ALLNODES, NODEPID )
    call relcube( cube(i) )
end do
stop
end
```

Appendix D: FORTRAN Ring Example Node Code Modifications

```
parameter ( NUMCUBES = 2, INTERCUBE = 50 )
integer*4 cubemap
logical cmpname
character*15 cube(NUMCUBES)
character*15 name, mycube
character*1 digit(0:9)
data digit/'0','1',..., '9'/
:
do i = 1, NUMCUBES
    cube(i) = "cube" // digit(i)
end do
call initcubecomm()
name = mycube()
:
if ((mynode() .eq. 0) .and. cmpname(name, cube(1))) then
:
else
    if (mynode() .eq. 1) then
        if (name .eq. cube(1)) then
            call crecv(NODETYPE, msgbuff, MAXMSGSIZE )
            rcnt = infocount()
            node = cubemap( cube(2), 1 )
            call csend(INTERCUBE, msgbuff, rcnt, node, 0)
            call crecv(INTERCUBE, msgbuff, MAXMSGSIZE)
            rcnt = infocount()
            call csend(NODETYPE, msgbuff, rcnt, nextnode, nextpid)
        else
            call crecv(INTERCUBE, msgbuff, MAXMSGSIZE)
            rcnt = infocount()
            call csend(NODETYPE, msgbuff, rcnt, nextnode, nextpid)
            call crecv(NODETYPE, msgbuff, MAXMSGSIZE)
            rcnt = infocount()
            node = cubemap( cube(1), 1 )
            call csend(INTERCUBE, msgbuff, rcnt, node, 0)
        endif
    else
        call crecv(NODETYPE, msgbuff, MAXMSGSIZE)
        rcnt = infocount()
        call csend(NODETYPE, msgbuff, rcnt, nextnode, nextpid)
    endif
endif
```

